# 3.1 Widget简介

## 3.1.1 概念

在前面的介绍中，我们知道在Flutter中几乎所有的对象都是一个Widget。与原生开发中“控件”不同的是，Flutter中的Widget的概念更广泛，它不仅可以表示UI元素，也可以表示一些功能性的组件如：用于手势检测的 GestureDetector widget、用于APP主题数据传递的Theme等等，而原生开发中的控件通常只是指UI元素。在后面的内容中，我们在描述UI元素时可能会用到“控件”、“组件”这样的概念，读者心里需要知道他们就是widget，只是在不同场景的不同表述而已。由于Flutter主要就是用于构建用户界面的，所以，在大多数时候，读者可以认为widget就是一个控件，不必纠结于概念。

## 3.1.2 Widget与Element

在Flutter中，Widget的功能是“描述一个UI元素的配置数据”，它就是说，Widget其实并不是表示最终绘制在设备屏幕上的显示元素，而它只是描述显示元素的一个配置数据。

实际上，Flutter中真正代表屏幕上显示元素的类是Element，也就是说Widget只是描述Element的配置数据！有关Element的详细介绍我们将在本书后面的高级部分深入介绍，现在，读者只需要知道：**Widget只是UI元素的一个配置数据，并且一个Widget可以对应多个Element**。这是因为同一个Widget对象可以被添加到UI树的不同部分，而真正渲染时，UI树的每一个Element节点都会对应一个Widget对象。总结一下：

* Widget实际上就是Element的配置数据，Widget树实际上是一个配置树，而真正的UI渲染树是由Element构成；不过，由于Element是通过Widget生成的，所以它们之间有对应关系，在大多数场景，我们可以宽泛地认为Widget树就是指UI控件树或UI渲染树。
* 一个Widget对象可以对应多个Element对象。这很好理解，根据同一份配置（Widget），可以创建多个实例（Element）。

读者应该将这两点牢记在心中。

## 3.1.3 Widget主要接口

我们先来看一下Widget类的声明：

@immutable  
abstract class Widget extends DiagnosticableTree {  
 const Widget({ this.key });  
 final Key key;  
   
 @protected  
 Element createElement();  
  
 @override  
 String toStringShort() {  
 return key == null ? '$runtimeType' : '$runtimeType-$key';  
 }  
  
 @override  
 void debugFillProperties(DiagnosticPropertiesBuilder properties) {  
 super.debugFillProperties(properties);  
 properties.defaultDiagnosticsTreeStyle = DiagnosticsTreeStyle.dense;  
 }  
   
 static bool canUpdate(Widget oldWidget, Widget newWidget) {  
 return oldWidget.runtimeType == newWidget.runtimeType  
 && oldWidget.key == newWidget.key;  
 }  
}

* Widget类继承自DiagnosticableTree，DiagnosticableTree即“诊断树”，主要作用是提供调试信息。
* Key: 这个key属性类似于React/Vue中的key，主要的作用是决定是否在下一次build时复用旧的widget，决定的条件在canUpdate()方法中。
* createElement()：正如前文所述“一个Widget可以对应多个Element”；Flutter Framework在构建UI树时，会先调用此方法生成对应节点的Element对象。此方法是Flutter Framework隐式调用的，在我们开发过程中基本不会调用到。
* debugFillProperties(...) 复写父类的方法，主要是设置诊断树的一些特性。
* canUpdate(...)是一个静态方法，它主要用于在Widget树重新build时复用旧的widget，其实具体来说，应该是：是否用新的Widget对象去更新旧UI树上所对应的Element对象的配置；通过其源码我们可以看到，只要newWidget与oldWidget的runtimeType和key同时相等时就会用newWidget去更新Element对象的配置，否则就会创建新的Element。

有关Key和Widget复用的细节将会在本书后面高级部分深入讨论，读者现在只需知道，为Widget显式添加key的话可能（但不一定）会使UI在重新构建时变的高效，读者目前可以先忽略此参数。本书后面的示例中，只会在构建列表项UI时会显式指定Key。

另外Widget类本身是一个抽象类，其中最核心的就是定义了createElement()接口，在Flutter开发中，我们一般都不用直接继承Widget类来实现一个新组件，相反，我们通常会通过继承StatelessWidget或StatefulWidget来间接继承Widget类来实现。StatelessWidget和StatefulWidget都是直接继承自Widget类，而这两个类也正是Flutter中非常重要的两个抽象类，它们引入了两种Widget模型，接下来我们将重点介绍一下这两个类。

## 3.1.4 StatelessWidget

在之前的章节中，我们已经简单介绍过StatelessWidget，StatelessWidget相对比较简单，它继承自Widget类，重写了createElement()方法：

@override  
StatelessElement createElement() => new StatelessElement(this);

StatelessElement 间接继承自Element类，与StatelessWidget相对应（作为其配置数据）。

StatelessWidget用于不需要维护状态的场景，它通常在build方法中通过嵌套其它Widget来构建UI，在构建过程中会递归的构建其嵌套的Widget。我们看一个简单的例子：

class Echo extends StatelessWidget {  
 const Echo({  
 Key key,   
 @required this.text,  
 this.backgroundColor:Colors.grey,  
 }):super(key:key);  
   
 final String text;  
 final Color backgroundColor;  
  
 @override  
 Widget build(BuildContext context) {  
 return Center(  
 child: Container(  
 color: backgroundColor,  
 child: Text(text),  
 ),  
 );  
 }  
}

上面的代码，实现了一个回显字符串的Echo widget。

按照惯例，widget的构造函数参数应使用命名参数，命名参数中的必要参数要添加@required标注，这样有利于静态代码分析器进行检查。另外，在继承widget时，第一个参数通常应该是Key，另外，如果Widget需要接收子Widget，那么child或children参数通常应被放在参数列表的最后。同样是按照惯例，Widget的属性应尽可能的被声明为final，防止被意外改变。

然后我们可以通过如下方式使用它：

Widget build(BuildContext context) {  
 return Echo(text: "hello world");  
}

运行后效果如图3-1所示：

图3-1

### Context

build方法有一个context参数，它是BuildContext类的一个实例，表示当前widget在widget树中的上下文，每一个widget都会对应一个context对象（因为每一个widget都是widget树上的一个节点）。实际上，context是当前widget在widget树中位置中执行”相关操作“的一个句柄，比如它提供了从当前widget开始向上遍历widget树以及按照widget类型查找父级widget的方法。下面是在子树中获取父级widget的一个示例：

class ContextRoute extends StatelessWidget {  
 @override  
 Widget build(BuildContext context) {  
 return Scaffold(  
 appBar: AppBar(  
 title: Text("Context测试"),  
 ),  
 body: Container(  
 child: Builder(builder: (context) {  
 // 在Widget树中向上查找最近的父级`Scaffold` widget  
 Scaffold scaffold = context.ancestorWidgetOfExactType(Scaffold);  
 // 直接返回 AppBar的title， 此处实际上是Text("Context测试")  
 return (scaffold.appBar as AppBar).title;  
 }),  
 ),  
 );  
 }  
}

运行后效果如图3-1-1所示：

图3-1-1

**注意**：对于BuildContext读者现在可以先作了解，随着本书后面内容的展开，也会用到Context的一些方法，读者可以通过具体的场景对其有个直观的认识。关于BuildContext更多的内容，我们也将在后面高级部分再深入介绍。

## 3.1.5 StatefulWidget

和StatelessWidget一样，StatefulWidget也是继承自Widget类，并重写了createElement()方法，不同的是返回的Element 对象并不相同；另外StatefulWidget类中添加了一个新的接口createState()。

下面我们看看StatefulWidget的类定义：

abstract class StatefulWidget extends Widget {  
 const StatefulWidget({ Key key }) : super(key: key);  
   
 @override  
 StatefulElement createElement() => new StatefulElement(this);  
   
 @protected  
 State createState();  
}

* StatefulElement 间接继承自Element类，与StatefulWidget相对应（作为其配置数据）。StatefulElement中可能会多次调用createState()来创建状态(State)对象。
* createState() 用于创建和Stateful widget相关的状态，它在Stateful widget的生命周期中可能会被多次调用。例如，当一个Stateful widget同时插入到widget树的多个位置时，Flutter framework就会调用该方法为每一个位置生成一个独立的State实例，其实，本质上就是一个StatefulElement对应一个State实例。
* 在本书中经常会出现“树”的概念，在不同的场景可能指不同的意思，在说“widget树”时它可以指widget结构树，但由于widget与Element有对应关系（一可能对多），在有些场景（Flutter的SDK文档中）也代指“UI树”的意思。而在stateful widget中，State对象也和StatefulElement具有对应关系（一对一），所以在Flutter的SDK文档中，可以经常看到“从树中移除State对象”或“插入State对象到树中”这样的描述。其实，无论哪种描述，其意思都是在描述“一棵构成用户界面的节点元素的树”，读者不必纠结于这些概念，还是那句话“得其神，忘其形”，因此，本书中出现的各种“树”，如果没有特别说明，读者都可抽象的认为它是“一棵构成用户界面的节点元素的树”。

## 3.1.6 State

一个StatefulWidget类会对应一个State类，State表示与其对应的StatefulWidget要维护的状态，State中的保存的状态信息可以：

1. 在widget 构建时可以被同步读取。
2. 在widget生命周期中可以被改变，当State被改变时，可以手动调用其setState()方法通知Flutter framework状态发生改变，Flutter framework在收到消息后，会重新调用其build方法重新构建widget树，从而达到更新UI的目的。

State中有两个常用属性：

1. widget，它表示与该State实例关联的widget实例，由Flutter framework动态设置。注意，这种关联并非永久的，因为在应用声明周期中，UI树上的某一个节点的widget实例在重新构建时可能会变化，但State实例只会在第一次插入到树中时被创建，当在重新构建时，如果widget被修改了，Flutter framework会动态设置State.widget为新的widget实例。
2. context。StatefulWidget对应的BuildContext，作用同StatelessWidget的BuildContext。

#### State生命周期

理解State的生命周期对flutter开发非常重要，为了加深读者印象，本节我们通过一个实例来演示一下State的生命周期。在接下来的示例中，我们实现一个计数器widget，点击它可以使计数器加1，由于要保存计数器的数值状态，所以我们应继承StatefulWidget，代码如下：

class CounterWidget extends StatefulWidget {  
 const CounterWidget({  
 Key key,  
 this.initValue: 0  
 });  
  
 final int initValue;  
  
 @override  
 \_CounterWidgetState createState() => new \_CounterWidgetState();  
}

CounterWidget接收一个initValue整型参数，它表示计数器的初始值。下面我们看一下State的代码：

class \_CounterWidgetState extends State<CounterWidget> {   
 int \_counter;  
  
 @override  
 void initState() {  
 super.initState();  
 //初始化状态   
 \_counter=widget.initValue;  
 print("initState");  
 }  
  
 @override  
 Widget build(BuildContext context) {  
 print("build");  
 return Scaffold(  
 body: Center(  
 child: FlatButton(  
 child: Text('$\_counter'),  
 //点击后计数器自增  
 onPressed:()=>setState(()=> ++\_counter,  
 ),  
 ),  
 ),  
 );  
 }  
  
 @override  
 void didUpdateWidget(CounterWidget oldWidget) {  
 super.didUpdateWidget(oldWidget);  
 print("didUpdateWidget");  
 }  
  
 @override  
 void deactivate() {  
 super.deactivate();  
 print("deactive");  
 }  
  
 @override  
 void dispose() {  
 super.dispose();  
 print("dispose");  
 }  
  
 @override  
 void reassemble() {  
 super.reassemble();  
 print("reassemble");  
 }  
  
 @override  
 void didChangeDependencies() {  
 super.didChangeDependencies();  
 print("didChangeDependencies");  
 }  
  
}

接下来，我们创建一个新路由，在新路由中，我们只显示一个CounterWidget：

Widget build(BuildContext context) {  
 return CounterWidget();  
}

我们运行应用并打开该路由页面，在新路由页打开后，屏幕中央就会出现一个数字0，然后控制台日志输出：

I/flutter ( 5436): initState  
I/flutter ( 5436): didChangeDependencies  
I/flutter ( 5436): build

可以看到，在StatefulWidget插入到Widget树时首先initState方法会被调用。

然后我们点击⚡️按钮热重载，控制台输出日志如下：

I/flutter ( 5436): reassemble  
I/flutter ( 5436): didUpdateWidget  
I/flutter ( 5436): build

可以看到此时initState 和didChangeDependencies都没有被调用，而此时didUpdateWidget被调用。

接下来，我们在widget树中移除CounterWidget，将路由build方法改为：

Widget build(BuildContext context) {  
 //移除计数器   
 //return CounterWidget();  
 //随便返回一个Text()  
 return Text("xxx");  
}

然后热重载，日志如下：

I/flutter ( 5436): reassemble  
I/flutter ( 5436): deactive  
I/flutter ( 5436): dispose

我们可以看到，在CounterWidget从widget树中移除时，deactive和dispose会依次被调用。

下面我们来看看各个回调函数：

* initState：当Widget第一次插入到Widget树时会被调用，对于每一个State对象，Flutter framework只会调用一次该回调，所以，通常在该回调中做一些一次性的操作，如状态初始化、订阅子树的事件通知等。不能在该回调中调用BuildContext.inheritFromWidgetOfExactType（该方法用于在Widget树上获取离当前widget最近的一个父级InheritFromWidget，关于InheritedWidget我们将在后面章节介绍），原因是在初始化完成后，Widget树中的InheritFromWidget也可能会发生变化，所以正确的做法应该在在build（）方法或didChangeDependencies()中调用它。
* didChangeDependencies()：当State对象的依赖发生变化时会被调用；例如：在之前build() 中包含了一个InheritedWidget，然后在之后的build() 中InheritedWidget发生了变化，那么此时InheritedWidget的子widget的didChangeDependencies()回调都会被调用。典型的场景是当系统语言Locale或应用主题改变时，Flutter framework会通知widget调用此回调。
* build()：此回调读者现在应该已经相当熟悉了，它主要是用于构建Widget子树的，会在如下场景被调用：
  1. 在调用initState()之后。
  2. 在调用didUpdateWidget()之后。
  3. 在调用setState()之后。
  4. 在调用didChangeDependencies()之后。
  5. 在State对象从树中一个位置移除后（会调用deactivate）又重新插入到树的其它位置之后。
* reassemble()：此回调是专门为了开发调试而提供的，在热重载(hot reload)时会被调用，此回调在Release模式下永远不会被调用。
* didUpdateWidget()：在widget重新构建时，Flutter framework会调用Widget.canUpdate来检测Widget树中同一位置的新旧节点，然后决定是否需要更新，如果Widget.canUpdate返回true则会调用此回调。正如之前所述，Widget.canUpdate会在新旧widget的key和runtimeType同时相等时会返回true，也就是说在在新旧widget的key和runtimeType同时相等时didUpdateWidget()就会被调用。
* deactivate()：当State对象从树中被移除时，会调用此回调。在一些场景下，Flutter framework会将State对象重新插到树中，如包含此State对象的子树在树的一个位置移动到另一个位置时（可以通过GlobalKey来实现）。如果移除后没有重新插入到树中则紧接着会调用dispose()方法。
* dispose()：当State对象从树中被永久移除时调用；通常在此回调中释放资源。

StatefulWidget生命周期如图3-2所示：

图3-2

**注意**：在继承StatefulWidget重写其方法时，对于包含@mustCallSuper标注的父类方法，都要在子类方法中先调用父类方法。

### 为什么要将build方法放在State中，而不是放在StatefulWidget中？

现在，我们回答之前提出的问题，为什么build()方法放在State（而不是StatefulWidget）中 ？这主要是为了提高开发的灵活性。如果将build()方法在StatefulWidget中则会有两个问题：

* 状态访问不便。
* 试想一下，如果我们的StatefulWidget有很多状态，而每次状态改变都要调用build方法，由于状态是保存在State中的，如果build方法在StatefulWidget中，那么build方法和状态分别在两个类中，那么构建时读取状态将会很不方便！试想一下，如果真的将build方法放在StatefulWidget中的话，由于构建用户界面过程需要依赖State，所以build方法将必须加一个State参数，大概是下面这样：
* Widget build(BuildContext context, State state){  
   //state.counter  
   ...  
   }
* 这样的话就只能将State的所有状态声明为公开的状态，这样才能在State类外部访问状态！但是，将状态设置为公开后，状态将不再具有私密性，这就会导致对状态的修改将会变的不可控。但如果将build()方法放在State中的话，构建过程不仅可以直接访问状态，而且也无需公开私有状态，这会非常方便。
* 继承StatefulWidget不便。
* 例如，Flutter中有一个动画widget的基类AnimatedWidget，它继承自StatefulWidget类。AnimatedWidget中引入了一个抽象方法build(BuildContext context)，继承自AnimatedWidget的动画widget都要实现这个build方法。现在设想一下，如果StatefulWidget 类中已经有了一个build方法，正如上面所述，此时build方法需要接收一个state对象，这就意味着AnimatedWidget必须将自己的State对象(记为\_animatedWidgetState)提供给其子类，因为子类需要在其build方法中调用父类的build方法，代码可能如下：
* class MyAnimationWidget extends AnimatedWidget{  
   @override  
   Widget build(BuildContext context, State state){  
   //由于子类要用到AnimatedWidget的状态对象\_animatedWidgetState，  
   //所以AnimatedWidget必须通过某种方式将其状态对象\_animatedWidgetState  
   //暴露给其子类   
   super.build(context, \_animatedWidgetState)  
   }  
  }
* 这样很显然是不合理的，因为
  1. AnimatedWidget的状态对象是AnimatedWidget内部实现细节，不应该暴露给外部。
  2. 如果要将父类状态暴露给子类，那么必须得有一种传递机制，而做这一套传递机制是无意义的，因为父子类之间状态的传递和子类本身逻辑是无关的。

综上所述，可以发现，对于StatefulWidget，将build方法放在State中，可以给开发带来很大的灵活性。

## 3.1.7 在Widget树中获取State对象

由于StatefulWidget的的具体逻辑都在其State中，所以很多时候，我们需要获取StatefulWidget对应的State对象来调用一些方法，比如Scaffold组件对应的状态类ScaffoldState中就定义了打开SnackBar(路由页底部提示条)的方法。我们有两种方法在子widget树中获取父级StatefulWidget的State对象。

### 通过Context获取

context对象有一个ancestorStateOfType(TypeMatcher)方法，该方法可以从当前节点沿着widget树向上查找指定类型的StatefulWidget对应的State对象。下面是实现打开SnackBar的示例：

Scaffold(  
 appBar: AppBar(  
 title: Text("子树中获取State对象"),  
 ),  
 body: Center(  
 child: Builder(builder: (context) {  
 return RaisedButton(  
 onPressed: () {  
 // 查找父级最近的Scaffold对应的ScaffoldState对象  
 ScaffoldState \_state = context.ancestorStateOfType(  
 TypeMatcher<ScaffoldState>());  
 //调用ScaffoldState的showSnackBar来弹出SnackBar  
 \_state.showSnackBar(  
 SnackBar(  
 content: Text("我是SnackBar"),  
 ),  
 );  
 },  
 child: Text("显示SnackBar"),  
 );  
 }),  
 ),  
);

上面示例运行后，点击”显示SnackBar“，效果如图3-1-2所示：

图3-1-2

一般来说，如果StatefulWidget的状态是私有的（不应该向外部暴露），那么我们代码中就不应该去直接获取其State对象；如果StatefulWidget的状态是希望暴露出的（通常还有一些组件的操作方法），我们则可以去直接获取其State对象。但是通过context.ancestorStateOfType获取StatefulWidget的状态的方法是通用的，我们并不能在语法层面指定StatefulWidget的状态是否私有，所以在Flutter开发中便有了一个默认的约定：如果StatefulWidget的状态是希望暴露出的，应当在StatefulWidget中提供一个of静态方法来获取其State对象，开发者便可直接通过该方法来获取；如果State不希望暴露，则不提供of方法。这个约定在Flutter SDK里随处可见。所以，上面示例中的Scaffold也提供了一个of方法，我们其实是可以直接调用它的：

...//省略无关代码  
// 直接通过of静态方法来获取ScaffoldState   
ScaffoldState \_state=Scaffold.of(context);   
\_state.showSnackBar(  
 SnackBar(  
 content: Text("我是SnackBar"),  
 ),  
);

### 通过GlobalKey

Flutter还有一种通用的获取State对象的方法——通过GlobalKey来获取！ 步骤分两步：

1. 给目标StatefulWidget添加GlobalKey。

* //定义一个globalKey, 由于GlobalKey要保持全局唯一性，我们使用静态变量存储  
  static GlobalKey<ScaffoldState> \_globalKey= GlobalKey();  
  ...  
  Scaffold(  
   key: \_globalKey , //设置key  
   ...   
  )

1. 通过GlobalKey来获取State对象

* \_globalKey.currentState.openDrawer()

GlobalKey是Flutter提供的一种在整个APP中引用element的机制。如果一个widget设置了GlobalKey，那么我们便可以通过globalKey.currentWidget获得该widget对象、globalKey.currentElement来获得widget对应的element对象，如果当前widget是StatefulWidget，则可以通过globalKey.currentState来获得该widget对应的state对象。

注意：使用GlobalKey开销较大，如果有其他可选方案，应尽量避免使用它。另外同一个GlobalKey在整个widget树中必须是唯一的，不能重复。

## 3.1.8 Flutter SDK内置组件库介绍

Flutter提供了一套丰富、强大的基础组件，在基础组件库之上Flutter又提供了一套Material风格（Android默认的视觉风格）和一套Cupertino风格（iOS视觉风格）的组件库。要使用基础组件库，需要先导入：

import 'package:flutter/widgets.dart';

下面我们介绍一下常用的组件。

#### 基础组件

* [Text](https://docs.flutter.io/flutter/widgets/Text-class.html)：该组件可让您创建一个带格式的文本。
* [Row](https://docs.flutter.io/flutter/widgets/Row-class.html)、 [Column](https://docs.flutter.io/flutter/widgets/Column-class.html)： 这些具有弹性空间的布局类Widget可让您在水平（Row）和垂直（Column）方向上创建灵活的布局。其设计是基于Web开发中的Flexbox布局模型。
* [Stack](https://docs.flutter.io/flutter/widgets/Stack-class.html)： 取代线性布局 (译者语：和Android中的FrameLayout相似)，[Stack](https://docs.flutter.io/flutter/widgets/Stack-class.html)允许子 widget 堆叠， 你可以使用 [Positioned](https://docs.flutter.io/flutter/widgets/Positioned-class.html) 来定位他们相对于Stack的上下左右四条边的位置。Stacks是基于Web开发中的绝对定位（absolute positioning )布局模型设计的。
* [Container](https://docs.flutter.io/flutter/widgets/Container-class.html)： [Container](https://docs.flutter.io/flutter/widgets/Container-class.html) 可让您创建矩形视觉元素。container 可以装饰一个[BoxDecoration](https://docs.flutter.io/flutter/painting/BoxDecoration-class.html), 如 background、一个边框、或者一个阴影。 [Container](https://docs.flutter.io/flutter/widgets/Container-class.html) 也可以具有边距（margins）、填充(padding)和应用于其大小的约束(constraints)。另外， [Container](https://docs.flutter.io/flutter/widgets/Container-class.html)可以使用矩阵在三维空间中对其进行变换。

#### Material组件

Flutter提供了一套丰富的Material组件，它可以帮助我们构建遵循Material Design设计规范的应用程序。Material应用程序以[MaterialApp](https://docs.flutter.io/flutter/material/MaterialApp-class.html) 组件开始， 该组件在应用程序的根部创建了一些必要的组件，比如Theme组件，它用于配置应用的主题。 是否使用[MaterialApp](https://docs.flutter.io/flutter/material/MaterialApp-class.html)完全是可选的，但是使用它是一个很好的做法。在之前的示例中，我们已经使用过多个Material 组件了，如：Scaffold、AppBar、FlatButton等。要使用Material 组件，需要先引入它：

import 'package:flutter/material.dart';

#### Cupertino组件

Flutter也提供了一套丰富的Cupertino风格的组件，尽管目前还没有Material 组件那么丰富，但是它仍在不断的完善中。值得一提的是在Material 组件库中有一些组件可以根据实际运行平台来切换表现风格，比如MaterialPageRoute，在路由切换时，如果是Android系统，它将会使用Android系统默认的页面切换动画(从底向上)；如果是iOS系统，它会使用iOS系统默认的页面切换动画（从右向左）。由于在前面的示例中还没有Cupertino组件的示例，下面我们实现一个简单的Cupertino组件风格的页面：

//导入cupertino widget库  
import 'package:flutter/cupertino.dart';  
  
class CupertinoTestRoute extends StatelessWidget {  
 @override  
 Widget build(BuildContext context) {  
 return CupertinoPageScaffold(  
 navigationBar: CupertinoNavigationBar(  
 middle: Text("Cupertino Demo"),  
 ),  
 child: Center(  
 child: CupertinoButton(  
 color: CupertinoColors.activeBlue,  
 child: Text("Press"),  
 onPressed: () {}  
 ),  
 ),  
 );  
 }  
}

下面（图3-3）是在iPhoneX上页面效果截图：

图3-3

### 关于示例

本章后面章节的示例中会使用一些布局类组件，如Scaffold、Row、Column等，这些组件将在后面“布局类组件”一章中详细介绍，读者可以先不用关注。

### 总结

Flutter提供了丰富的组件，在实际的开发中你可以根据需要随意使用它们，而不必担心引入过多组件库会让你的应用安装包变大，这不是web开发，dart在编译时只会编译你使用了的代码。由于Material和Cupertino都是在基础组件库之上的，所以如果我们的应用中引入了这两者之一，则不需要再引入flutter/widgets.dart了，因为它们内部已经引入过了。